OOProg - Zusammenfassung

Pointer

Dynamischer Speicher (S. 98)

```
int* i1 = new int
delete i1;

int* i2 = new int[5];
delete i2;        // Falsch! würde nur 1. Element löschen
delete[] i2;      // Richtig! Löscht ganzes Array
```

void-Pointer (S. 100)

```
int i = 17;
void* anyPtr = &i;
```

Funktionspointer (S. 101)

Funktion	Wirkung
void err1 (char);*	Prototyp
*void *err2 (char*);*	Prototyp, void*-Rückgabewert
*void (*fErr) (char*);*	Zeiger auf Funktion mit char*-Parameter

const Pointer (S. 99)

Pointer	Wirkung
char p1*	non-const Pointer auf non-const-Objekt
char const p2*	const-Pointer auf non-const-Objekt
const char p3*	non-const-Pointer auf const-Objekt
const char const p4*	const-Pointer auf const-Objekt

Namespace (S. 124)

Deklaration: *namespace MyLib1 { int i; void foo(); }*
Aufruf: *MyLib1::foo();*
using namespace Mylib; foo();

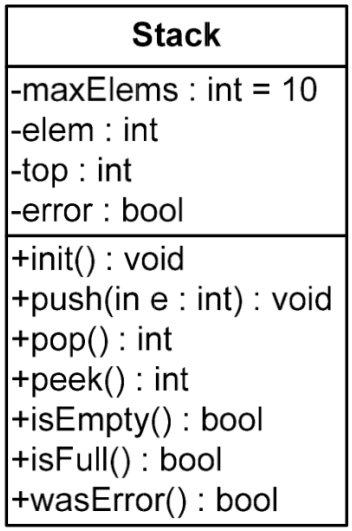
- Namespaces sind offen – d.h. es ist jederzeit möglich, neue Variablen, Funktionen etc. hinzuzufügen (gleich wie bei Deklaration)
- Nameless namespace: *namespace { void foo(); }* ➔ Systemweit eindeutig Aufruf via *::foo();*
- NIE using Namespace in Header!

Typen

- *volatile:* verhindert „aggressive“ Optimierungen durch Compiler. Wird oft bei Embedded Systems für Register verwendet.
- *extern:* Um in einer anderen Datei deklarierte Variablen einbinden
- *register:* Hinweis/Bitte an Compiler: Variable wenn möglich in Register legen
- *static:* in Datenbereich statt auf Stack gespeichert. Automatisch mit 0 initialisiert. Wird nie gelöscht: 1. Fkt-Aufruf – Wert setzen – 2. Fkt-Aufruf – Wert noch vorhanden! Kann in C++ durch nameless namespaces ersetzt werden (bevorzugt)
- *mutable:* kann von const-Methoden geändert werden

Klassen

Definition (S. 117, S. 170ff)




```
class Stack
{
   public:
        void init();
        void push(int e);
        int pop();
        int peek() const;
        bool isEmpty() const;
        bool isFull() const;
        bool wasError() const;
   private:
        enum {maxElems = 10};
        int elem[maxElems];
        int top;
        mutable bool error;
        // mutable: auch const-Methoden können dieses Attribut setzen
};
```

Ctor / Dtor (S. 186 / S. 195)

Initialisierungsliste: (bevorzugt, weil schneller)
Dummy::Dummy(string newName, int newCode):name(newName),code(newCode) {}

Overloading: mehrere Arten zum Initialisieren ermöglichen; mehrere Ctors können public, protected, private erstellt werden um die Verwendbarkeit einzuschränken (z.B. nur private kann auf eine bestimmte Art initialisieren)

Copy Constructor: immer erstellen! Wird benötigt um ein neues Objekt mit bestehendem zu initialisieren.

Shallow Copy: Wird standardmässig erstellt; Kopiert alle vorhandenen Werte 1:1
Deep Copy: Wird benötigt, wenn Objekte auf dem Heap sind, muss selbst erstellt werden

Destruktor: *virtual ~Dummy();* - im Code: *Dummy::~Dummy()*

Templates

Verwendung: Gleiche Klasse / Funktion für verschiedene Typen nötig.

Funktions-Templates (S. 230)

Definition des Templates mit: *template<class ElemType>*
Anwendung:

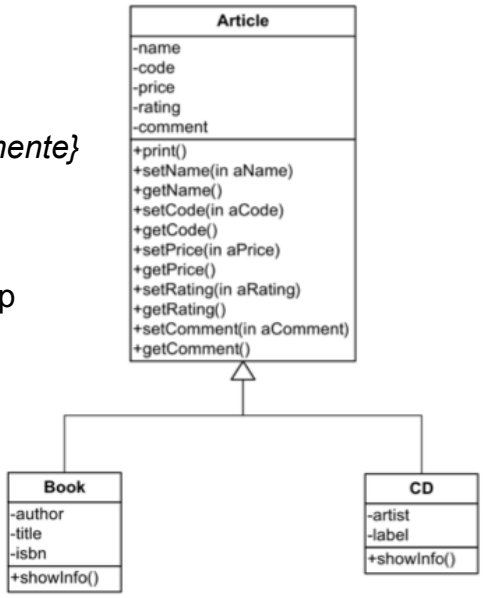
```
ElemType minimum(ElemType values[], int length)  // ElemType statt normalem Typ
int i = minimum<int>(ivalues, size);              // in <> den Typ angeben
```

Klassen-Templates (S. 234)

Typen oder Konstanten zur Parametrisierung der Klasse
Definition*: template <class ElemType, int size=100>*
Anwendung: *Stack<int, 10> s;* oder *Stack<int> s;* (→ Defaultwert 100)

Vererbung (S. 254)

class Article { ... }
class Book : public Article { Ctor, Dtor, neue Elemente}
class CD : public Article { ... }



- abgeleitete Klassen immer in neue .h / .cpp
- Anwendung bei is-a Beziehung:
  Book is an Article, CD is an Article

Zugriffsschutz (S. 178 / S. 256)

Schlüsselwort	Sichtbar in eigener Klasse	Sichtbar in Subklasse	Sichtbar von aussen
public	ja	ja	ja
protected	ja	ja	nein
private	ja	nein	nein

- **friend:** jeder friend darf auf alle Elemente zugreifen! - Schlechtes Design!
  Anwendung: unten in Klasse: *friend class Y; friend void foo();*

Polymorphismus (S. 275)

Polymorphismus bedeutet eine dynamische Bindung, d.h. während der Laufzeit wird entschieden, welchen Typ eine Variable hat und welche Methoden aufgerufen werden.

Virtuelle Elementfunktionen (S. 276)

Schlüsselwort virtual nicht nötig, aber übersichtlicher.

```
class A { virtual foo(); }
class B : public A { virtual foo(); }          // Überschreibt A::foo virtuell
```

Anwendung:

```
A* object;
object[0] = new A; object[0]->foo();       // A::foo wird aufgerufen
object[1] = new B; object[1]->foo();       // B::foo wird aufgerufen
```

Abstrakte Basisklassen (S. 285)

- Basisklassen legen Gemeinsamkeiten („Protokoll“) fest
- Abstrakte Funktion: *virtual void print() = 0;*
- 1 abstrakte (rein virtuelle, nicht definierte) Funktion → ganze Klasse ist abstrakt
- Kann erst durch Vererbung nutzbar gemacht werden.

Mehrfachvererbung (S. 288)

- eine Klasse von mehreren Basisklassen ableiten
- Anwendung: *class ClassC : public ClassA, public Class B { ... }*
- Mehrdeutige (in beiden Basisklassen existierende) Funktionen müssen mit Gültigkeitsbereich aufgerufen werden: *ClassA::print(); ClassB::print();*
- Besser: neue Funktion einfügen: *virtual void print() const { ClassA::print(); };*

- Virtuelle Basisklasse: *class ClassB : virtual public Class A*
- löst Zweideutigkeiten: es werden nur Verweise auf Basisklasse kopiert.

Exceptionhandling (S. 311)

Try / Catch: Auf Exceptions achten

```
try {
        foo();
}
catch (const ExceptionClass& exc) { /* do something */ }  // fängt ExceptionClass
catch (...) { /* do something else */ }                    // fängt alle Exceptions
```

Throw: Exception auslösen (S. 312)

```
throw „Fehler!“;        // Wirft eine Beschreibung → Schlecht: Typen nutzen
throw myException;      // Wirft eine Exception vom Typ myException
throw;                  // Wirft eine bestehende Exception weiter (nur in
                        // ExceptionHandler anwenden!)
```

Bei einem throw wird das Programm abgebrochen und im ExceptionHandler fortgesetzt. Alle lokalen Daten werden zerstört.

Spezifikation von Ausnahmen

- für jede Funktion festgelegt werden, welche Exception auftreten kann
- z.B. bei Stack: *void push() throw(overflow_error);*

Vordefinierte Ausnahmeklassen (S. 313)

Basisklasse: *exception*
Laufzeitfehler: nicht vorhersehbar
Logische Fehler: Fehler im Programmablauf

Eigene Exceptions am besten von *exception* ableiten!